MSO Lab Exercise III: Implementation of the NS ticket machine

Dennis Brunek 5988489
Victor van Weelden 6240577

Section 1: The current solution

In this section we will review the current solution for the ticket machine system on various factors.

What works well:

- Storing ticket data: The way of storing the ticket data using enumerables is a good solution for it. Because of this it is relatively easy to add new options in the future.

What could be improved:

- The way of calculating the price is very inflexible: Because of how the class PricingTable works the UI class needs to do the price calculation in a very specific way.
- The discount should be determined by the discount card: In the current solution the discount is determined by what the user selects and not by the discount card itself. This also causes the system to be inflexible to new discounts.
- Storing tariefeenheden: The current way that the tariefeenheden are stored is very inflexible and adding a new station would require a lot of new code to be written.
- You can buy a ticket from a station to the same station: In the current solution it is possible to buy a ticket from a station to the same station which should not be possible.
- The absence of a printer class and printer method: In the current solution there is nothing to represent the functionality of printing the tickets.

Easy to add changes/ requirements:

- New options in ticket: Because of the way the UIInfo class is made it is relatively easy to add new options to a ticket.

Changes/ requirements that are difficult to add:

- Adding new products: Because of abstract classes not being used adding a new product (think of a weekcard or something like that) would require a lot of new code and if statements to get it to work.
- Adding new payment methods: Similarly to adding a new product a new payment method would also be difficult because of the lack of a abstract class system.

- Changing price calculation: The current way of calculating the price is very propiatery and changing the prices or changing the way of calculating the price would require changing a lot of code.

Bad code smells:

- Low cohesion in UI class: In the current solution the UI class handles a lot of things which causes very low cohesion.
- Strong coupling between UI and PricingTable: Although the way these classes are coupled is not necessarily strong coupling it is still strong coupling because of the propiatery way the price is calculated using different columns to indicate the height of the price together with the amount of tariefeenheden. Calculating the price correctly requires the UI class to determine the amount of columns in a very specific way.
- Strong coupling between UI and UIInfo: The UI class depends on the UIInfo class at various locations. The UI.paymentmethod uses the UIInfo elements at the switches.

Section 2: Reflection on previous design

In this section we will reflect on the differences of our design and our implementation of our design. Our initial design can be found in appendix 1.

In this assignment we were relatively successful in implementing our design. The biggest change we made is that in our original design we had a PaymentMehod interface which had two subclasses ICoinMachine and ICard and in our implemented design we choose not to implement the PaymentMethod interface. We made this decision because we realized this class would not simplify the system and would not make it easier to implement a new payment method in the future. In Sale we can switch between the payment methods with one simple switch statement and adding a PaymentMethod interface would still make this switch statement necessary. Because of this removal Sale has slightly different methods: PayByCard and PayCash are no longer needed because they are included in the Pay method.

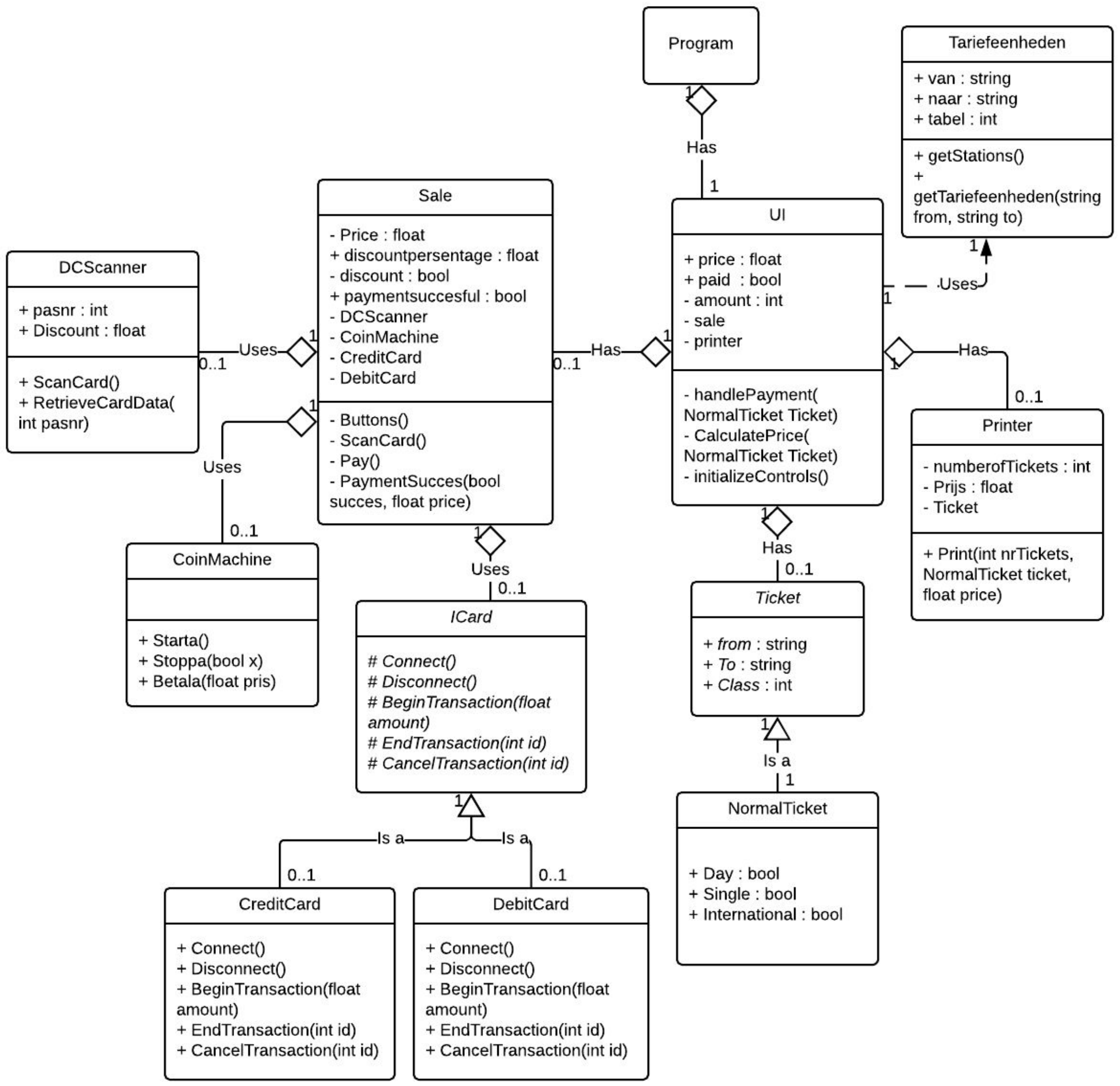
We also changed a lot of class and method names in our implemented design, but almost all the classes from our design are still there, just with a different name. The biggest name change is TicketHandler class that is now called UI, because next to being a controller class it also makes the UI. Also DiscountCardScanner is now called DCScanner. Also a lot methods have different names or multiple methods have been implemented in one method. The best example of this is the handlePayment method which has the functionalities of the CalculatePrice, PrintTicket and StartSale methods of our initial design. We did this because creating three separate methods which all have two or three lines of code does not make a lot of sense and because it is a controller class. It makes sense it has a method which controls multiple important aspects of the whole process. In the Sale class multiple methods have also been changed. PaymentMethodSelector, PayByCard and PayCash are not all one method called Pay, because the functionality of switching between payment methods and calling the needed classes can easily be done in one method. The Sale class now also has a method which creates a new form called Buttons. This method creates our overlay were the user chooses the payment method and chooses if he/she has a discount card to scan.

In our current implementation there are multiple classes which were not in our original design. The class Tariefeenheden is added because in our program (more of a prototype but that is the assignment) we do not use a database to get the tariefeenheden data. That is why this class which mimics a database exists. Also Program which has the static Main method is new because there needs to be a class with a static Main method which starts the UI class.

What have you learned about your original design?

We learned from this assignment that our original design was mostly well thought out and a useful guideline to implement our system. We also found out when using the given program that that give program had a lot of flaws and a lot of classes that were poorly implemented. A good example is the UI class where multiple important functionalities happened in one method.

Appendix 1: UML class diagram of our initial design


Program
Has
1
1
Tariefeenheden
+ van : string
+ naar : string
+ tabel : int
+ getStations()
+ getTariefeenheden(string from, string to)
Sale
- Price : float
+ discountpersentage : float
- discount : bool
+ paymentsuccesful : bool
- DCScanner
- CoinMachine
- CreditCard
- DebitCard
- Buttons()
- ScanCard()
- Pay()
- PaymentSucces(bool succes, float price)
UI
+ price : float
+ paid : bool
- amount : int
- sale
- printer
- handlePayment(NormalTicket Ticket)
- CalculatePrice(NormalTicket Ticket)
- initializeControls()
Uses
DCScanner
+ pasnr : int
+ Discount : float
+ ScanCard()
+ RetrieveCardData(int pasnr)
Uses
0..1
1
Has
0..1
1
Has
1
0..1
Printer
- numberofTickets : int
- Prijs : float
- Ticket
+ Print(int nrTickets, NormalTicket ticket, float price)
Uses
0..1
CoinMachine
+ Starta()
+ Stoppa(bool x)
+ Betala(float pris)
Uses
0..1
ICard
# Connect()
# Disconnect()
# BeginTransaction(float amount)
# EndTransaction(int id)
# CancelTransaction(int id)
Has
0..1
Ticket
+ from : string
+ To : string
+ Class : int
Is a
1
NormalTicket
+ Day : bool
+ Single : bool
+ International : bool
Is a
Is a
0..1
0..1
CreditCard
+ Connect()
+ Disconnect()
+ BeginTransaction(float amount)
+ EndTransaction(int id)
+ CancelTransaction(int id)
DebitCard
+ Connect()
+ Disconnect()
+ BeginTransaction(float amount)
+ EndTransaction(int id)
+ CancelTransaction(int id)